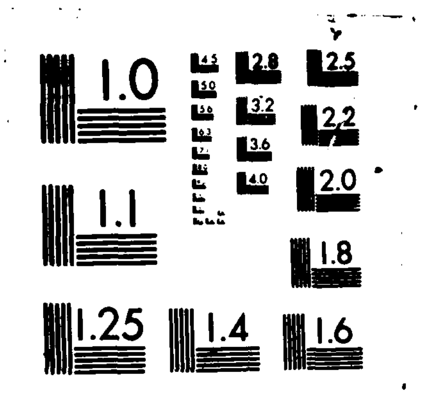
AD-A183 631 TOWARD A REQUIREMENTS APPRENTICE: ON THE BOUNDARY BETWEEN INFORMAL AND FO..(U) MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE L.. C RICH ET AL. 1/1

UNCLASSIFIED JUL 86 AI-M-907 N00014-85-K-0124 F/G 12/9 NL

END
9-87
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DTIC FILE COPY (1)

AD-A183 631

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER A.I. Memo 907	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Towards a Requirements Apprentice: On the Boundary Between Informal and Formal Specifications		5. TYPE OF REPORT & PERIOD COVERED A.I. Memo 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Charles Rich & Richard C. Waters		8. CONTRACT OR GRANT NUMBER(s) N00014-85-K-0124
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE July, 1986 13. NUMBER OF PAGES 25
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited.

DTIC
ELECTE
AUG 24 1987
S D
E

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Programmer's Apprentice	Cliches
Knowledge Acquisition	Informality
Requirements	Specification

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Requirements acquisition is one of the most important and least well supported parts of the software development process. The Requirements Apprentice (RA) will assist a human analyst in the creation and modification of software requirements. Unlike current requirements analysis tools, which assume a formal description language, the focus of the RA is on the boundary between informal and formal specifications. The RA is intended to support the
(continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 | UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT CONTINUED

earliest phases of creating a requirement, in which incompleteness, ambiguity, and contradiction are inevitable features.

From an artificial intelligence persoective, the central problem the RA faces is one of knowledge acquisition. It has to develop a coherent internal representation from an initial set of disorganized statements. To do so, the RA will rely on a variety of techniques, including dependency-directed reasoning, hybrid knowledge representation, and the reuse of common forms (cliches).

The Requirements Apprentice is being developed in the context of the Programmer's Apprentice project, whose overall goal is the creation of an intelligent assistant for all aspects of software development.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 907 July, 1986

Toward a Requirements Apprentice: On the Boundary Between Informal and Formal Specifications

by

Charles Rich & Richard C. Waters

Abstract

Requirements acquisition is one of the most important and least well supported parts of the software development process. The Requirements Apprentice (RA) will assist a human analyst in the creation and modification of software requirements. Unlike current requirements analysis tools, which assume a formal description language, the focus of the RA is on the boundary between informal and formal specifications. The RA is intended to support the earliest phases of creating a requirement, in which *incompleteness*, *ambiguity*, and *contradiction* are inevitable features.

From an artificial intelligence perspective, the central problem the RA faces is one of *knowledge acquisition*. It has to develop a coherent internal representation from an initial set of disorganized statements. To do so, the RA will rely on a variety of techniques, including dependency-directed reasoning, hybrid knowledge representation, and the reuse of common forms (*clichés*).

The Requirements Apprentice is being developed in the context of the Programmer's Apprentice project, whose overall goal is the creation of an intelligent assistant for all aspects of software development.

Adapted from a proposal to the National Science Foundation.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by International Business Machines and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

1. Introduction

The Programmer's Apprentice project uses the domain of programming as a vehicle for studying (and attempting to duplicate) human problem solving skills. Recognizing that it will be a long time before it is possible to fully duplicate human abilities in this domain, the near-term goal of the project is the development of a system, called the Programmer's Apprentice, which provides intelligent assistance in various phases of the programming task.

Viewed at the highest level, software development is a process that begins with the desires of an end user and ends with a program that can be executed on a machine. The first step of this process is traditionally called requirements acquisition, while the last step is called implementation. Figure 1 shows how the current and proposed demonstration systems in the Programmer's Apprentice project support these activities.

```
USER ++++++++++++++>-------------<+++++++++++++<============= MACHINE
      Requirements                      New         KBEmacs
       Apprentice                  Demonstration
```

Figure 1: Spectrum of software development activities.

To date, most of the research in the Programmer's Apprentice project has focused on program implementation. This has resulted in the creation of a working demonstration system called the Knowledge-Based Editor in Emacs (KBEmacs) [62,63,64]. The principal benefit of KBEmacs is that it allows a programmer to construct a program rapidly and reliably by combining algorithmic fragments stored in a library. An additional benefit of the knowledge-based editing approach (not fully exploited in KBEmacs) is that it provides a basis for intelligent program modification and maintenance. The principal limitations of KBEmacs are that it has a narrow view of the programming process and weak reasoning abilities.

Currently, much of the effort in the Programmer's Apprentice project is being directed toward the construction of a new demonstration system which will have increased reasoning abilities and which will therefore be able to assist in a greater portion of the programming process. In particular, the new system will be able to detect many kinds of design errors which KBEmacs cannot detect. The new system will also be able to deduce implicit design decisions which follow from explicit decisions made by the user.

This proposal is to begin development of a Requirements Apprentice (RA), which will assist an analyst in the creation and modification of software requirements. The RA will eventually link up with the other parts of the Programmer's Apprentice, which are growing "up" from the implementation end of the spectrum. In the meantime, research on the RA establishes a second beachhead from which to attack the problem of automating the programming process. Requirements acquisition is an opportune place for such a beachhead because, like implementation (and unlike the middle parts of the programming process), it is constrained by contact with a real-world boundary.

Research on requirements acquisition is valuable for two reasons. From the perspective of artificial intelligence, it is a good domain in which to pursue fundamental questions related to knowledge acquisition. From the perspective of software engineering, requirements acquisition is perhaps the most crucial part of the software process. Studies (e.g. [7]) indicate that errors in requirements are more costly than any other kind of error. Furthermore, requirements acquisition is not currently well supported by software tools.

The body of this proposal begins with a discussion of the requirements acquisition task. This discussion delineates the focus of research on the RA. Section 3 defines the fundamental artificial intelligence and software engineering research issues, namely: understanding the process by which informal descriptions evolve into formal specifications and codifying the knowledge which is used in software requirements. Section 4 is a scenario which illustrates the capabilities of the RA. Section 5 reviews the accomplishments of the Programmer's Apprentice project to date with emphasis on those elements which contribute most directly to the RA, namely: hybrid knowledge representation and reasoning, codification of programming knowledge, and principles for implementing intelligent computer assistants.

2. Focusing the Research

It is useful to distinguish different phases in the requirements acquisition process. The earliest phase usually takes the form of a "skull session", whose goal is achieving a consensus among a group of users about what they want. The requirements analyst's main role in this phase relies on his personal skills as a facilitator, perhaps with the aid of a team debriefing methodology, such as WISDM [34] or JAD [9]. The end product of this phase is typically an informal requirement.

Figure 2 (taken from [2]) shows an example of the informal requirement for a university library database. This particular requirement was used as a benchmark for comparing a number of different specification tools at the Second and Third Workshops on Software Specification and Design [2,24]. It is also used in this proposal as a basis for the scenario in Section 4.

Most work on software requirements tools (e.g. [5,11,12,17,21,35]) focuses on what is usually called the *validation* phase. The main goal of this part of the process is to increase confidence that a given requirement actually corresponds to the end user's desires. In current research approaches, this is achieved by applying simulation, symbolic execution, and various kinds of analysis to a formal specification. This work does not, however, address the key question of how a formal specification is constructed in the first place. For example, Kemmerer [21] begins by simply exhibiting the translation of the informal requirement of Figure 2 into the formal specification shown in Figure 3 (taken from [21]).

The focus of the RA is on bridging the gap between informal and formal specification. This is a crucial area of lack in the current state of the art. For example, it was reported at the Second Workshop on Software Specification and Design ([2], p. 107) that some of the greatest problems stemmed from "the process of completing the system analysis work needed to translate the informal specification into the appropriate input for the tools".

A second reason for focusing on the transition from informal to formal is that it brings up fundamental issues in artificial intelligence (see Section 3). As elsewhere in the Programmer's Apprentice project, there is an opportunity here both to apply current artificial intelligence

Consider a university library database. There are two types of users: normal borrowers and users with library staff status. The database transactions are:

(1) Check out a copy of a book.
(2) Return a copy of a book.
(3) Add a copy of a book to the library.
(4) Remove a copy of a book from the library.
(5) Remove all copies of a book from the library.
(6) Get a list of titles of books in the library by a particular author.
(7) Find out what books are currently checked out by a particular borrower.
(8) Find out what borrower last checked out a particular copy of a book.

These transactions have the following restrictions:

R1 - A copy of a book may be added or removed from the library only be someone with library staff status.
R2 - Library staff status is also required to find out which borrower last checked out a copy of a book.
R3 - A normal borrower may find out only what books he or she has checked out. However, a user with library staff status may find out what books are checked out by any borrower.

The requirements that the database must satisfy at all times are:

G1 - All copies in the library must be checked out or available for check out.
G2 - No copy of a book may be both checked out and available for check out.
G3 - A borrower may not have more than a given number of books checked out at any one time.
G4 - A borrower may not have more than one copy of the same book checked out at one time.

Figure 2: Example of an informal requirement (copyright (c) IEEE 1985).

techniques to a software engineering problem and to use a software engineering problem to drive further research in artificial intelligence.

Another key aspect of the RA is that it will contain an extensive library of knowledge about the particular domain of the requirement to be constructed. This is to be contrasted with, for example, a tool for the symbolic execution of a formal requirements language. Such a tool can do a lot to help identify problems with what is in a given requirement. However, it is not in a position to say very much about what might be missing from the requirement. Having specific knowledge about what should be in the requirements associated with a particular domain makes it possible for a tool like the RA to critique what is not in a requirement as well as what is in the requirement.

The current research with goals most similar to the RA is the KATE [13] system. Fickas has proposed an interactive system which will provide assistance over the entire requirements acquisition process. To make this feasible, Fickas intends to rely heavily on exploiting a particular example domain (conference planning).

Interaction with the RA

Figure 4 shows the role of the RA in relation to other agents involved in the software process. Note that the RA does not interact directly with an end user, but is an assistant to the requirements analyst. The RA also serves as a bridge between the requirements analyst and the system designer.

```
End User <-----> Analyst  <----->  RA  <----->  System Designer
```

Figure 4: Role of the RA.

The main benefit of excluding direct interaction with the end user is that it avoids having to deal with the complexity of natural language input. Free-form natural language input would be essential for interaction with a naive end user. An analyst, however, should have no trouble using a more restrictive command language. Natural language understanding is a major research area in its own right, which is, by and large, independent of the central issues in building the RA.

Output of the RA

In current practice, the end product of requirements acquisition is typically a single written document which is produced by the analyst and used both by the end user (for validation) and by the system designer (as the starting point for design). Using the RA, the essential end product of the requirements acquisition process is a machine-manipulable representation of the requirement inside the RA. In the long term, this internal representation will be accessed directly by other tools and components of the Programmer's Apprentice. In the short term, this information will be used to answer queries and to generate various documents for the requirements analyst, the end user, and the system designer. An advantage of the RA approach is that different organizations of the information can be produced, tailored to the different needs of the end user, the analyst, and the designer. Examples of the kinds of documents generated by the RA are shown in the scenario below.

```
Specification Library
LEVEL Top_Level

TYPE
   User,
   Book,
   Book_Title,
   Book_Author,
   Book_Collection = Set Of Book,
   Titles = Set Of Book_Title,
   Natural = T' i:Integer (i≥0)

CONSTANT
   Title(Book):Book_Title,
   Author(Book):Book_Author,
   Library_Staff(User):Boolean,
   Book_Limit:Natural,
   Copy_Of(B1:Book,B2:Book):Boolean =
        Author(B1) = Author(B2)
      & Title(B1) = Title(B2)

VARIABLE
   Library:Book_Collection,
   Checked_Out(Book):Boolean,
   Responsible(Book):User,
   Number_Books(User):Natural,
   Never_Out(Book):Boolean,

   User_Result:User,
   Book_Result:Book_Collection,
   Title_Result:Titles

DEFINE
   Available(B:Book):Boolean =
      B ∈ Library & ~Checked_Out(B),
   Checked_Out_To(U:User,B:Book):Boolean =
        Checked_Out(B)
      & Responsible(B)=U

CRITERION
     ∀ b:Book(b ∈ Library →
                 (Checked_Out(b) & ~Available(b)
               |  ~Checked_Out(b) & Available(b)))
   & ∀ u:User(Number_Books(u) ≤ Book_Limit)
   & ∀ u:User,b1,b2:Book(
         Checked_Out_To(u,b1)
       & Checked_Out_To(u,b2)
       & Copy_Of(b1,b2)
          → b1=b2)
INITIAL
     Library = Empty
   & ∀ u:User (Number_Books(u) = 0)
   & ∀ b:Book (~Checked_Out(b))
```

Figure 3: Formal Ina Jo[1] requirement corresponding to Figure 2 (copyright (c) IEEE 1985).

```
TRANSFORM Check_Out(U:User,B:Book) External
 Effect
  (if  Available(B)
    & Number_Books(U) < Book_Limit
    & ∀ B1:Book (Checked_Out_To(U,B1) → ~Copy_Of(B,B1))
       then ∀ U1:User (N"Number_Books(U1) =
              (if U1=U
                  then Number_Books(U) + 1
                  else Number_Books(U1)))
          & ∀ B1:Book (
              (if B1=B
                  then  N"Checked_Out(B)
                      & N"Responsible(B)=U
                      & ~N"Never_Out(B)
                  else  NC"(Checked_Out(B1), Responsible(B1),
                               Never_Out(B1))))
       else NC"(Number_Books,Checked_Out,Responsible,Never_

TRANSFORM Return(B:Book) External
 Effect
  (if Checked_Out(B)
      then  ∀ B1:Book (N"Checked_Out(B1) =
               (if B=B1
                   then False
                   else Checked_Out(B1)))
          & ∀ U1:User (N"Number_Books(U1) =
               (if U1=Responsible(B)
                   then Number_Books(U1) -1
                   else Number_Books(U1)))
      else NC"(Checked_Out,Number_Books))

TRANSFORM Add_A_Book(U:User,B:Book) External
 Effect
  (if  Library_Staff(U)
    & B ∉ Library
      then  N"Library = Library ∪ {B}
          & ∀ B1:Book (
               N"Checked_Out(B1) =
                 (if B=B1
                     then False
                     else Checked_Out(B1))
             & N"Never_Out(B1) =
                 (if B=B1
                     then True
                     else Never_Out(B1)))
      else NC"(Library,Checked_Out,Responsible,Never_Out))
TRANSFORM Remove_A_Book(U:User,B:Book) External
 Effect
  (if  Library_Staff(U)
    & Available(B)
       then N"Library = Library ~~ {B}
       else NC"(Library))
TRANSFORM Last_Responsible(U:User,B:Book) External
 Effect
  (if  Library_Staff(U)
    & B ∈ Library
    & ~Never_Out(B)
       then N"User_Result = Responsible(B)
       else NC"(User_Result))

TRANSFORM What_Checked_Out(Requester,Whom:User) External
 Effect
  (if (Library_Staff(Requester) | Requester=Whom)
     then ∀ B1:Book (
              Checked_Out_To(Whom,B1) & B1 ∈ N"Book_Result
            | ~Checked_Out_To(Whom,B1) & B1 ∉ N"Book_Result)
     else NC"(Book_Result))

TRANSFORM Titles_By_Author(By_Whom:Book_Author) External
 Effect
  N"Title_Result = {T1:Book_Title ( ∃ B1:Book (
                     Author(B1)=By_Whom & Title(B1)=T1))}

END Top_Level
END Library
```

1. Trademark of System Development Corporation, a Burroughs Company.

The documents generated by the RA could take many forms. They could be textual, rendered in a formal specification language, or diagrammatic. In the near term, effort will focus on the producing more or less traditional textual presentations of requirements. Since many software projects are contractually obligated to provide documents of this form, automatically generating (and re-generating) such documents will be a valuable near-term feature of the RA.

It is important not to focus too early in the development of the RA on designing a new formal specification or diagrammatic language. As with natural language understanding, it is not because these areas are unimportant, but rather because they are largely orthogonal to many of the key issues underlying the RA, and therefore can be temporarily side-stepped. Experience with other parts of the Programmer's Apprentice has demonstrated the benefits of concentrating first on designing an internal representation (see the Plan Calculus [49,51]) that is well suited for automated reasoning and other manipulations.

It is useful, however, as a check on the semantic adequacy of an internal representation, to demonstrate the ability to produce output in an existing language. (In the case of the Plan Calculus, we demonstrated the production of source code in Lisp and Ada.) A research milestone will therefore be to produce the formal specification shown in Figure 3 from an interaction similar to the scenario in Section 4.

3. Fundamental Issues

Research on the RA brings up two fundamental issues in knowledge acquisition. The first issue is *informality* and the process by which informal descriptions evolve into formal specifications. The second issue is the role and specific content of prior knowledge of the common structures (*clichés*) of a domain.

In the area of knowledge acquisition, the work most similar to the RA has been concerned with providing automated assistance for acquiring new rules for expert systems. Most systems, such as Teiresias [10] and Seek [28], are limited to fairly simple well-formedness and consistency checking. Other systems, such as KLAUS [18] and ROGET [6] provide for the acquisition of new concepts and vocabulary.

Informality

On the issue of informality, the RA continues in the tradition of the SAFE project [3]. Balzer, Goldman and Wile were the first to argue that informality is an inevitable (and ultimately desirable) feature of the specification process. They began by studying actual natural language software specifications, cataloging the kinds of informality they found. At the end of the project, the SAFE prototype system succeeded in automatically producing a formal specification (in the language AP2) from a pre-parsed informal natural language specification for a number of examples.

The following is a list of general features that characterize informal communication between a speaker (e.g. an end user) and hearer (e.g. an analyst). These features are based on the discussion in [3] and an initial study of the informal requirement in Figure 2.

Abbreviation — Special terms (jargon) are used. The hearer is assumed to have a large amount of specific knowledge which explains the terms.

Ambiguity — Statements can be interpreted in several different ways. The hearer has to disambiguate these statements based on the surrounding context.

Poor Ordering — Statements are presented in the order they occur to the speaker, rather than in an order that would be convenient for the hearer. The hearer needs to hold many questions in abeyance until later statements answer them.

Incompleteness — Aspects of the description are left out. The hearer has to fill in these gaps by using his own knowledge or asking questions.

Contradiction — Statements which are true in the main are liable to be contradictory in detail. This reflects the fact that the speaker has not thought things out completely.

Inaccuracy — For a variety of reasons, some of the statements are simply wrong.

These kinds of informality are not a matter of the speaker being lazy or incompetent. Informality is an essential part of the human thought process. It is part of a powerful *debugging* strategy for dealing with complexity, which shows up in many problem solving domains: Start with an almost-right description and then incrementally modify it until it is acceptable [32]. Thus, having the RA deal with informality is not just a question of being user friendly — it is a fundamental prerequisite.

One of the goals of research on the RA is to elaborate the initial characterization of informality given above, and to develop strategies and heuristics for removing these features from informal requirements.

The RA will differ from SAFE in several respects. First, the interface to the RA further separates natural language understanding from the essential informality issues. Although parentheses were added manually to the English input to avoid parsing difficulties, SAFE still attempted to deal with a number of other natural language phenomena, such as pronouns, which are better dealt with in other research.

Second, although the SAFE work points to the importance of domain knowledge in resolving informality, it lacks the notion of clichés as a way of representing, organizing, and applying this knowledge. Finally, SAFE is an automatic batch system, whereas the RA is an interactive assistant.

Clichés

Expert engineers rarely construct complex artifacts (automobiles, electronic circuits, or requirements specifications) by starting from first principles. Rather, they bring to the task their previous experience, in the form of knowledge of the commonly occurring structures (combinations of the primitives) in the domain. The term *cliché* is used here to refer to these commonly occurring structures. In normal usage, the word cliché has a pejorative sound that connotes overuse and a lack of creativity. However, in the context of engineering problem solving, this kind of reuse is a positive feature.

Knowledge of the relevant clichés is essential for effective communication on any topic. There is ample evidence that, in general, it is difficult, if not impossible, to acquire new knowledge unless one already has a large amount of relevant old knowledge. Imagine trying to communicate the requirements for an inventory control system to a person who knows nothing about either information systems or inventories.

Notions similar to the cliché idea appear in software engineering in the work of Arango and Freeman [1] (domain models), Harandi and Young [19] (design templates), and Lavi [22] (generic models); and in artificial intelligence in the work of Minsky [25,26] (frames, concept germs), Schank [30] (conceptual structures), and Chapman [38] (cognitive clichés).

Formally, a cliché consists of a set of *roles* embedded in an underlying *matrix*. The roles of a cliché are the parts that vary from one use of the cliché to the next. The matrix of the cliché contains both fixed elements of structure (parts that are present in every occurrence) and *constraints*. Constraints are used both to check that the parts that fill the roles in a particular occurrence are consistent, and also to compute parts to fill empty roles in a partially specified occurrence.

Requirements Clichés

A major goal of research on the RA is the codification of clichés in the domain of software requirements. This codification will include both clichés of broad applicability, like the three examples discussed below, and more specific clichés in several application areas. Such a taxonomy would be valuable even if it only existed as a textual handbook for use by a human analyst. An important benefit of orienting the RA around the use of clichés is that domain-specific knowledge can be provided as data, rather than built into the system. New domains can be covered by defining new clichés.

The importance of domain-specific knowledge is a theme which the RA shares with the PHI-NIX [4] and DRACO [27] projects. Neither of these projects, however, focuses on supporting informality.

Three examples of requirements clichés used in the RA scenario in Section 4 are: *repository*, *information system*, and *tracking system*. A repository is an entity in the physical world. The repository cliché has a number of roles including: the *items* which are stored in the repository, the *place* where the items are stored, the *staff* which manages the repository, and the *users* which utilize the repository. The basic function of a repository is to ensure that items which enter the repository will be available for later removal.

There are a variety of physical constraints which apply to repositories. For example, since each item has a physical existence, it can only be in one place at a time and therefore must either be in the repository or not.

There are several kinds of repositories. Simple repositories merely take in items and then give them out. A more complex kind of repository supports the lending of items, which are expected to be returned. Another dimension of variation concerns the items themselves. The items may be unrelated or they may be grouped into classes. Example repositories include: storage warehouses (simple repositories for unrelated items) grocery stores (simple repositories for items grouped in classes) and rental car agencies (lending repositories for items grouped in classes)

In contrast to the repository cliché, the information system cliché describes a class of programs rather than a class of physical objects. The intent of the information system cliché is to capture the commonality between programs such as personnel systems, bibliographic data bases, and inventory control systems.

The central role of an information system is the *information schema*. This is a meta-description which specifies the logical characteristics of the data to be stored. Since information system is a requirements cliché rather than an implementation cliché, these characteristics do not include how the data is physically organized in storage.

Other roles of an information system include: a set of *transactions* which can create/modify/delete the data, a set of *reports* which display parts of the data, *integrity constraints* on the data, a *staff* which manage the information system, and *users* which utilize the information system.

At a more detailed level, the information system cliché contains information about timing, security, error checking, and the like. For example, it has information about how to restrict access for different classes of users and how to check for and deal with errors in data entry.

A tracking system is a specialized kind of information system which keeps track of the state of a physical object. The roles of a tracking system are the same as the roles of an information system, with the addition of one new role: the *target* being tracked.

The target object is assumed to have a (possibly complex) state and to be subject to various physical operations which can modify this state. The information in the tracking system describes the state of the target object. The transactions modify this information to reflect changes in the target's state.

The main content of the tracking system cliché is a set of constraints which relate roles of the information system part of the cliché to the target role. For example, the constraints can be used to derive the information schema from a description of the possible states of the target. Similarly, an appropriate set of transactions can be derived from the operations applicable to the target. In addition, any physical constraints on the target become integrity constraints on the information system.

There are several kinds of tracking systems. A tracking system may follow several targets instead of just one. A tracking system may keep a history of past states of the target. A tracking system may operate based on direct observations of the state of the target or based on observations of operations on the target. Finally, a tracking system may participate in controlling the operations on the target, rather than merely observing them. Example tracking systems include: aircraft tracking systems (which track multiple targets based on direct observations of their position) and inventory control systems (which track a repository based on observations of operations which modify its contents and often exercise some control over what can be given out to whom.)

4. Scenario

To make the proposed capabilities of the RA more concrete, the following presents a scenario of how the system will interact with a requirements analyst. The scenario is based on the library database example published in [2] (reproduced in Figure 2 above). This example has the virtue of being familiar enough to be easily understood by the general reader. It deals with a requirement for a university library data base system, which keeps track of who has borrowed which books. (This scenario is based on a thesis proposal by Reubenstein [47]).

Before the scenario begins, it is important to set the scene by describing what the RA is expected to know beforehand. We assume that the RA knows nothing about libraries *per se*. The RA does, however, know a considerable amount about the general kinds of systems of which the library data base is an instance. Specifically, the RA knows about repositories, information systems, and tracking systems, as described in the previous section.

In a given situation, the RA could have either more or less relevant knowledge beforehand. The level of knowledge in this scenario was chosen in order to illustrate a useful middle ground. On one hand, if the RA knew significantly more than what is postulated above (e.g., if it had a cliché for a library database), then the interaction between the analyst and the RA would look very much like using an application generator. Although the RA would, of course, be very useful in this mode, such a scenario would not do a good job of showing the way the RA is typically intended to be used.

On the other hand, if the RA were missing any of the clichés discussed above, then the analyst would have to say too much. In particular, he would have to describe the missing clichés. Although the RA would still be usable in this mode, it is unlikely that any tool which seeks to dramatically improve the productivity and reliability of the requirements acquisition process can do so without a rich store of prior knowledge.

Since the user interface to the RA has not been fully designed, the interactions in this scenario are intended to illustrate the major features of this interface without being overly specific about details. For instance, input typed by the analyst will be shown in simple English (after the prompt ">"). However, as discussed earlier, the RA will not support unrestricted natural language input. A stylized command language will be provided which supports the necessary semantic content illustrated in the scenario, but not the same degree of syntactic flexibility.

Finally, note that several errors have intentionally been introduced into what the analyst says in the scenario. The particular errors chosen may or may not appear plausible to particular readers. However, large numbers of errors are made during the creation of a typical requirement (most of which look pretty stupid in retrospect). The errors introduced here were chosen to illustrate the capabilities of the RA to detect and help correct errors in general.

Beginning the Requirement

With the commands shown below, an analyst begins the process of using the RA to construct a requirement. Note in the first command that new terms, such as the name LIBDB, are introduced in quotes. The second command gives the overall structure of the desired system. The third and fourth commands define the terms *library* and *copy of a book*.

```
>Begin a requirement for a system called "LIBDB".
>LIBDB is a tracking system which tracks a "library".
>A library is a repository for "copies of books".
>A copy of a book has the properties:
  title - a text string,
  author - a person's name,
  ISBN number - a unique alphanumeric key.
```

The net effect of a sequence of commands such as the one above is to augment the information contained in the RA's internal representation of the evolving requirement. This new information comes from three fundamentally different sources: explicit statements, clichés, and inferences.

After the four commands above, almost all of what the RA knows comes from the combination of the tracking system and repository clichés. In particular, an instance of the tracking system cliché is built with a library (an instance of the repository cliché) in the target role. Since the state of a repository is the collection of items it contains (in this case, copies of books), the constraints in the tracking system cliché are used to derive an information schema that provides fields for the three properties listed above for copies of books. Also based on the constraints in the tracking system cliché, an expectation is created within the RA that a set of transactions for LIBDB will be defined corresponding to the typical operations on a repository.

Note that the new terms LIBDB, library, and copy of a book are far from fully defined at this point. They are both incomplete and ambiguous. They are incomplete because many roles remain to be filled in. They are ambiguous because it is not yet clear which kind of tracking system LIBDB is or which kind of repository a library is. Since incompleteness and ambiguity are inevitable during the early stages of constructing a requirement, the RA refrains from complaining at this point. It accepts information and performs inferences on a "catch as catch can" basis. However, if requested, the RA can produce a list of currently unresolved issues (see Figure 5) and can guide the analyst in finishing the requirement.

Textual Displays

Output from the RA comes in two forms. First there are direct statements to the analyst, for example, describing a contradiction that has been discovered. The primary form of output, however, is textual displays generated from parts of the RA's internal representation of the evolving requirement. One kind of display corresponds to viewing sections of a requirements document. Other displays consist of various outlines and summaries.

The default response of the RA to commands from the analyst is to create a textual display which summarizes the major effect of the commands. The analyst can request the generation of other kinds of displays.

The state of the RA's default display after the four commands above is shown in Figure 5. The top part of the display shows the table of contents of the requirement as a whole. The bottom part of the display shows the purpose section which would be generated if a requirements document were to be created at this point in the scenario. The bottom of the purpose section summarizes, at an appropriately high level, the main points in the requirement which are currently unresolved.

```
                         Table of Contents
1.  Introduction
    1.1  Purpose
    1.2  Scope
    1.3  Definitions, Acronyms, and Abbreviations
    1.4  References
    1.5  Overview
2.  General Description
    2.1  Product Perspective
    2.2  Product Functions
    2.3  User Characteristics
    2.4  General Constraints
    2.5  Assumptions and Dependencies
3.  Specific Requirements
    3.1  Functional Requirements
         3.1.1  Transactions
         3.1.2  Reports
    3.2  External Interface Requirements
    3.3  Performance Requirements
    3.4  Design Constraints
    3.5  Attributes
         3.6.1 Security
         3.6.2 Error handling
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

1.1 Purpose

    The LIBDB system is a tracking system which tracks a library.  A
library is a repository for copies of books.
    LIBDB records information about the title, author, and ISBN number of
the copies of books in the library.  Transactions are supported for
tracking what happens to the repository.  Reports are provided for
obtaining information about the copies of books in the library.

Unresolved issues:
  What kind of tracking system is LIBDB?
  What kind of repository is a library?
  What transactions are supported?
  What reports are generated?
```

Figure 5: Textual Display Generated by the RA.

An important function of the displays is to focus attention. When the RA generates a display, it naturally focuses the analyst's attention on the information it contains. When the analyst requests a display, the RA can use the context of the display to disambiguate ensuing commands.

The structure of the requirement would vary based on the needs of the organization using the RA. (Figure 5 conforms to the ANSI/IEEE requirements document Standard 830-1984 [20].) The structure of the requirement also varies based on the top level clichés being used in the requirement. (In Figure 5 the choice of detail sections such as 3.1.1 are dictated by the tracking system cliché.)

Although the first part of the purpose section in Figure 5 is shown in connected English sentences, most of the output generated by the RA is expected to look more like the point-form listing of unresolved issues. As in the case of understanding free-flowing English input, the syntactic aspects of good English are not a key concern here. What is important is deciding *what* to say, for example, choosing the appropriate level of detail (as in [33]).

At one extreme, the RA could assume that the reader has a complete understanding of every cliché. At the other extreme, the RA could describe every cliché used in full detail. The first approach is unwarranted. The second approach would lead to a document which is too large and is likely to be too redundant with what the reader does know. As illustrated here, the RA will attempt to take a middle course, reminding the reader of the various clichés used, but not giving the complete details of each cliché unless asked.

Defining Transactions

The next set of commands begin the definition of the transactions to be supported by LIBDB. The analyst first directs the RA's attention to the topic of transactions by moving to the transactions subsection of the document. He then describes the *check out* transaction.

```
>Display the transactions subsection.
>The "check out" transaction tracks the removal of a copy of a book.
```

The key term in the second command is *removal*. Removal is one of the standard operations supported by a repository, i.e., taking an item out of the repository and giving it to a user of the repository. Constraints in the tracking system cliché have already generated the expectation of a corresponding transaction. This command gives this transaction a name (check out) and triggers its inclusion in the requirement. Another consequence of this command is that the RA infers that LIBDB is a tracking system based on observations of operations on the target, rather than on direct observations of its state.

The effect of the two commands above on the display is illustrated by Figure 6. Most of the information in Figure 6 comes from the repository cliché via the constraints in the tracking system cliché. The bottom of Figure 6 lists a number of issues which still need to be resolved.

```
3.1.1.1 Check Out
   The "check out" transaction tracks the removal of a copy of a book from
the library.
INPUTS: identifier of a copy of a book (ISBN number).
OUTPUTS: none.
PRECONDITIONS: The input must be in the roster of copies of books which
  are in the library.
EFFECT ON THE INFORMATION STORE: The input is removed from the roster of
  copies of books which are in the library.
UNUSUAL EVENTS: If the input is not in the roster of copies of books which
  are in the library, then the information system is inconsistent with the
  state of the repository.  A notation is made in the error log.  The
  inconsistency is alleviated by leaving the input out of the roster of
  copies of books which are in the library.
USAGE RESTRICTIONS: none.
Unresolved issues:
  Should historical record keeping be added?
  Should checking of user validity be added?
  Should checking of staff member validity be added?
```

Figure 6: Textual Display of the Check Out Transaction.

Fixing an Error

With the next set of commands, the analyst continues on the topic of transactions. Note the use of the term *inverse* to define the return transaction. In the repository cliché, the operation of adding an item into the repository is defined to be the inverse of removing an item. Therefore, the first command below triggers the generation of a transaction that tracks the addition of a book.

```
>The "return" transaction is the inverse of check out.
>The "add" transaction tracks the addition of a copy of a book.
Potential problem:
  The add transaction is exactly the same as the return transaction.
```

The command defining the add transaction provokes an immediate complaint from the RA (shown in italics), because it defines adding a copy of a book to be exactly the same as returning a copy of a book. This contradicts the RA's expectation that new terms should not be synonymous with old terms unless explicitly declared to be so. The analyst does not have to deal with this problem immediately — he could go on to other things leaving the problem pending. In this case, however, he decides to deal with the problem right away.

In order to begin the process of tracking down the source of the problem, the analyst asks the RA to present a trace of the chain of inference which led to the immediate manifestation of the problem. Many of the issues involved in tracking down the ultimate source of a bug in a requirement are similar to those of bug localization in programs [31,54]. For example, filtering has to be applied to restrict the set of all logically necessary premises to the likely candidates for retraction in the current context. In the explanation below, the RA does not include the fact that

LIBDB is a tracking system, because this is not likely to be retracted at this point.

```
>Explain the problem.
(1) The add transaction is exactly the same as the return transaction.
Because:
  (2) The add transaction tracks the addition of a copy of a book.
  (3) The return transaction tracks the addition of a copy of a book.
  Because:
    (4) The check out transaction tracks the removal of a copy of a book.
    (5) The return transaction is the inverse of the check out transaction.
```

Studying this explanation, the analyst realizes that he made an error earlier in the definition of check out. Checking out a book does not correspond to removing it from the repository, but rather to borrowing it from the repository. With the commands below, the analyst corrects the error b redefining the check out transaction.

```
>Retract (4).
>The check out transaction tracks the borrowing of a copy of a book.
Note: The return transaction now tracks the return of copy of a book
         to a lending repository.
```

Since statement (5) was not retracted, the return transaction is still the inverse of check out, which implies that the return transaction now tracks the return of a copy of a book. In addition, the RA has inferred that a library is a lending repository, which resolves an outstanding ambiguity.

Fixing a Conceptual Confusion

The scenario continues below with the definition of two transactions related to removing items from the library.

```
>The "remove" transaction tracks the removal of a copy of a book.
>The "remove all" transaction tracks the removal of every copy of a book.
Potential problem:
  The term every suggests that copies of books are members of classes.
  If this is not the case, then the remove all transaction is exactly
  the same as the remove transaction.
```

The second command above brings to the surface a conceptual confusion which has been buried in the requirement thus far. By using the word *every*, the analyst suggests that the *remove all* transaction corresponds to removing all instances of a class of items from the library. This is a standard operation on repositories which contain classes of items rather than unrelated items. However, the requirement thus far does not include a class/instance relationship for the library.

On seeing the RA's response, the analyst (assumedly in consultation with the end user) thinks a bit more about books and realizes that what were originally defined as the properties of a copy of a book should really be properties of a class called book. A copy of a book is then an instance of a particular class of books. He informs the RA below of this conceptual change. Among other things. this removes the last element of ambiguity as to what kind of repository a library is.

```
>Book is a class with properties previously defined for copy of a book.
>Redefine a copy of a book to be an instance of a book.
>A copy of a book has the property:
  copy number - a number unique within the class.
```

This conceptual reorganization is propagated by the RA throughout the requirement. One effect of the reorganization is to reveal that remove and remove all are indeed distinct transactions. In addition, there are a number of other changes. For example, the first input to the check out transaction is changed from an ISBN number to a pair of an ISBN number and a copy number.

Defining a Report

The next section of the scenario illustrates a different mode of interaction between the analyst and the RA, in which direct editing of the textual display is used for input. Direct editing has two advantages, both of which have been confirmed by experience with the similar interface to KBEmacs [46,62]. First, it is an essential escape mechanism, which makes it possible for the analyst to add information to the requirement that is beyond the RA's capability to understand. Second, even when information can be provided through the use of RA commands, it is often more convenient to provide it by direct editing.

```
>Display the reports subsection.
>Add a report called "books checked out to user".
```

In contrast to transactions, reports come in such a wide variety that the tracking system cliché does not have very much specific information about any individual report. As a result, the effect of the second command above is simply to insert a template of headings in a new subsection of the reports display. The analyst defines the new report by directly editing the textual display, supplying the information after each heading. The top part of Figure 7 shows the state of the display after the analyst has finished editing it (underlining indicates sections typed by the analyst).

When the analyst has completed his editing, the RA analyzes the result. The success of the analysis depends on the extent to which the RA has clichés which explain the terms used by the analyst. When terms are understood, the text is converted to an internal representation. When terms are not understood, the text is merely stored as is. In this case, the RA is assumed to understand all the important terms in Figure 7. For example, it knows what a due date is and that items lent from repositories often have due dates associated with them; it knows about sorting things by date; and it knows that having no items to report is a standard unusual event associated with reports. Given an understanding of these terms, the RA's analysis reveals a number of issues which still need to be resolved. These are summarized by the RA in the bottom part of Figure 7.

```
3.1.2.1 Books Checked Out To User
   The report "books checked out to user" lists all of the books lent to a
given user.
INPUTS: identifier of a user.
PRECONDITIONS: input must be a valid user.
REPORTED ITEMS: set of C: copy of a book, such that C is checked out to
   input.
REPORTED INFORMATION: title of C, author of C, copy number of C, due date
   of C.
SORT ORDER: by due date.
HEADING: "Books Checked Out To" input.
UNUSUAL EVENTS: If no items to report, print "No books borrowed".
USAGE RESTRICTIONS: .

   Unresolved issues:
      How is user identifier validity checked?
      What should be done if a user identifier is invalid?
```

Figure 7: Direct Editing of a Report Definition.

Following analysis, the RA also propagates the information it understands to other relevant parts of the requirement. For example, the information schema for LIBDB is extended to include due dates and a checked-out-to relation between copies of books and users. This in turn indicates that the check out transaction must take a user identifier as a second input.

Finishing the Requirement

With the addition of a number of commands similar to the ones above, the analyst could enter all of the information in the informal requirement in Figure 2. The analyst might then assert that the requirement was finished. This would cause the RA to check the requirement for completeness and to complain about a number of issues. For example, much more needs to be said about the users and staff and how they are identified. The RA would also complain that there is no transaction for entering a book into the library and therefore no way to initialize the system. Once all of these issues were resolved, the analyst could request that the RA produce a complete requirements document.

Observations on the Scenario

The scenario above illustrates that use of the RA can improve both productivity and the quality of the requirement produced. These benefits stem from three essential features of the RA, which act together synergistically: clichés, propagation of information, and contradiction detection.

Clichés directly improve productivity by allowing reuse of parts of requirements from project to project. Propagation of information improves productivity by allowing the analyst to provide each piece of information just once, at the point which is most convenient. The RA copies this information to other places where it is relevant.

Contradiction detection improves the quality of the final requirement. This is facilitated by the fact that most errors cause a number of contradictions — the RA only needs to be able to find one of them to recognize that there is an error. The use of clichés contributes to contradiction detection because of the large amount of predefined information, which is attached to them. Propagating information further increases the number of opportunities to find contradictions.

Two important facilities that are not illustrated in the scenario above are an interface for the analyst to use to familiarize himself with the available clichés and a mechanism for defining new clichés. The RA will have a "browsing" facility which will allow an analyst to inspect the cliché library. As in KBEmacs, a textual representation will be provided for clichés so that they can be easily inspected and defined. However, it should be noted that defining a major cliché, such as information system or repository, is a difficult task akin to writing the definitive paper on the subject. The typical analyst is expected to define only simple clichés, leaving the definition of major clichés to expert analysts who specialize in the construction of clichés.

5. Accomplishments to Date

Research on the RA will be pursued within the context of the Programmer's Apprentice project. Accomplishments of the project to date include theoretical work and demonstration systems in the areas of:

Representation of programming knowledge [48,49,52]
Automated reasoning techniques [43,51,53,55,56,57]
Knowledge-based program editing [62,63,64]
Principles for implementing intelligent computer assistants [39,46]
Automated program analysis [60,61,68]
Program translation [41,42,65]
Debugging [45,54]
Documentation [40,44,59,67]
Program testing [37]
Program transformation [58]

Although many extensions will be necessary, this work provides an intellectual platform upon which to build the RA. Three elements of the research to date that will contribute most directly toward building the RA are: a hybrid knowledge representation and reasoning system (Cake), codification of programming clichés, and principles for implementing intelligent computer assistants.

Hybrid Knowledge Representation and Reasoning

Historically, there have been two major approaches to knowledge representation and reasoning. One approach has emphasized the use of predicate calculus and general purpose theorem proving techniques. The other approach, partly in reaction to the difficulty of general purpose theorem proving, has emphasized special purpose representations and reasoning methods tailored to the structure of particular problem domains.

Experience in the Programmer's Apprentice project with reasoning about programs suggests that both types of techniques are needed. Special purpose representations and associated algorithms are essential in order to avoid the uncontrollable combinatorial explosions which often occur in predicate calculus based reasoning systems. On the other hand, predicate calculus reasoning is very valuable when used, under strict control, as the "glue" between inferences made in different special purpose representations. A hybrid knowledge representation and reasoning system, called Cake [43,51,53] has been implemented. it supports several different kinds of special purpose reasoning layered on top of a simple, general purpose, predicate calculus based reasoning system. Figure 8 shows the layers of the Cake system that are most relevant to the RA.

```
      FRAMES
       TYPES
      ALGEBRA
     EQUALITY
TRUTH MAINTENANCE
```

Figure 8: Layers of Cake.

The bottommost layer of Cake, truth maintenance, is essentially the boolean constraint propagation network from RUP [23]. It provides three principal facilities. First, it acts as a recording medium for dependencies, and thus supports retraction and explanation. (The explanation in the scenario using *Because* is a presentation of dependency information.) Second, it performs simple "one-step" deductions — specifically, unit propositional resolution. (This will provide the propagation of information illustrated in the scenario.) Third, the truth maintenance layer detects contradictions. Contradictions are represented explicitly in such a way that reasoning can continue with other information not involved in the contradiction. (This allows the RA to let the analyst postpone dealing with problems.)

The equality layer of Cake provides an incremental congruence closure facility, also taken from RUP. Given any two terms, the equality layer will determine whether they can be proved equal by substitution of equals using the set of currently true equalities between terms. (Reasoning about equality between transactions is invoked several times in the scenario to detect potential problems.)

The algebra layer of Cake is composed of special-purpose decision procedures for common algebraic properties of operators, such as commutativity, associativity, transitivity, inverses, and so on. These properties come up everywhere in formal modeling tasks. (In the scenario, the return transaction is defined as the inverse of check out.)

The types layer of Cake provides a full lattice of subtypes, with intersection, union and complement types. The notion of types is a basic facility used in all knowledge representation and formal specification systems. (At the end of the scenario, books are defined as types, of which a copy of a book is an instance.)

Finally, the frames layer, which is built using facilities from many of the layers below, supports the conventional frame notions of slots, instances, and inheritance. These facilities will be used in the RA as the basis for representing clichés and organizing the cliché library.

In the area of the applying knowledge representation techniques to software requirements, an important first step was the RML language of Greenspan [15]. Knowledge representation and requirements specification have a shared concern with modeling slices of the real world. Along with Mylopoulos and Borgida [8,16], Greenspan has begun the task of bringing together these two disciplines. The RA can be viewed as continuing this work into the inferencing, contradiction detection, and other dynamic issues not yet addressed in RML.

Programming Clichés

Study of the form and content of programming clichés is at the heart of the Programmer's Apprentice project. This aspect of the research has progressed in alternating cycles of codification and formalization, beginning with Waters' identification [60,61] of common loop forms, followed by Rich's formalization [48] of several hundred clichés in the area of manipulating symbolic data structures (sets, sequences, lists and graphs), and continuing with Wertheimer's codification [66] of the programming clichés used to build deduction-based programs, such as production systems, constraint propagation, GPS, and Prolog.

An important part of Rich's work was the development of organizing principles for a library of clichés using the notions of specialization, extension, and implementation. The KBEmacs system [62] demonstrates how clichés are used in program construction. Zelinka [68] has developed a system for automatically recognizing programming clichés, using graph parsing techniques developed by Brotsky [36].

Research on the Programmer's Apprentice has resulted in considerable experience representing and using clichés. Although requirements clichés are somewhat different in nature than programming clichés (they have more constraints and less fixed structure), it appears that the same principles [52] carry over. The formal representation must have enough expressive power to capture the variety of possible clichés. The properties of combinations of clichés must be easy to compute. There must be a sound semantic foundation to allow for formal verification of libraries of clichés. The representation must not be too closely tied to the syntax of any particular programming/specification language.

Intelligent Computer Assistants

When it is not possible to construct a fully automatic system for a task, it is, nevertheless, often possible to construct a system which can assist an expert in the task. In addition to yielding useful systems in the short run, the assistant approach can also provide important insights into how to construct a fully automatic system.

Work on KBEmacs and related systems has lead to the development of a set of design principles for intelligent computer assistants (see [39,46]). A computer assistant should be *non-invasive*: when not providing help, it should not present the user with constant reminders of its presence. A computer assistant should be *non-prescriptive*: it is up to the assistant to conform to the user's methods, not vice versa. A computer assistant should maintain *partial state*: the user may wish to be working on several aspects of the project at once; the system should not force him to finish one subproblem before beginning another. These principles are equally applicable to the RA, and are embodied in the scenario above.

The intelligent computer assistant approach taken in the RA is consistent with the approach recommended in the Knowledge-Based Software Assistant report [14]. In particular, the RA is based on the view that even if the software process cannot be totally automated at this time, it should be totally machine-mediated. The RA also assumes an evolutionary view of the software lifecycle. Some of the short term goals of the requirements facet of the Software Assistant are currently being worked on at Sanders Associates [29]. The RA begins to address the longer term goals laid out in the report.

References

[1] G. Arango and P. Freeman. "Modeling Knowledge for Software Development", *Proc. of Third Int. Workshop on Software Specification and Design*, London, UK, August, 1985, pp. 63-66.

[2] R. Babb, *et al.*, "Workshop on Models and Languages for Software Specification and Design", *IEEE Computer Magazine*, Vol. 18, No. 3, March, 1985, pp. 103-108.

[3] R. Balzer, N. Goldman and D. Wile, "Informality in Program Specifications", *IEEE Trans. on Software Eng.*, Vol. SE-4, No. 2, pp. 94-103, March, 1978.

[4] D.R. Barstow, "Domain-Specific Automatic Programming", *IEEE Trans. on Software Eng.*, Vol. 11, No. 11, pp. 1321-1336, Nov. 1985.

[5] T. Bell, D. Bixler, M. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering", *IEEE Trans. on Software Eng.*, Vol. 3, No. 1, January, 1977, pp. 49-59.

[6] J.S. Bennett, "A Knowledge-Based System for Acquiring the Conceptual Structure of a Diagnostic Expert System", *Journal of Automated Reasoning*, Vol. 1, No. 1, 1985.

[7] B.W. Boehm, "Verifying and Validating Software Requirements and Design Specifications", *IEEE Software Magazine*. January 1984, pp. 75-88

[8] A. Borgida, S. Greenspan, J. Mylopoulos, "Knowledge Representation as the Basis for Requirements Specifications", *IEEE Computer Magazine*, pp. 82-90, April 1985.

[9] A. Crawford, "Joint Application Design: A New Way to Design Systems", in *Guide International Proceedings*, Guide International Corporation, 1982

[10] R. Davis, "Applications of Meta-Level Knowledge to the Construction, Maintenance, and Use of Large Knowledge Bases", (Ph.D. Thesis), STAN-CS-76-564, Comp. Sci. Dept., Stanford U., 1976.

[11] A. Davis, T. Miller, E. Rhode, B. Taylor, "RLP: An Automated Tool for the Processing of Requirements", *COMPSAC 79*, Nov., 1979, pp. 289-299.

[12] M.S. Feather and P.E. London, "Implementing Specification Freedoms", *Science of Computer Programming* 2, pp. 91-131, 1982.

[13] S. Fickas, D. Laursen, J. Laursen, "A Knowledge-Based Software Specification Environment", *Workshop on Knowledge-Based Design*, Rutgers Univ., 1984.

[14] C. Green, D. Luckam, R. Balzer, T. Cheatham, C. Rich, "Report on a Knowledge-Based Software Assistant", Rome Air Development Center, Technical Report RADC-TR-83-195, August, 1983.

[15] S.J. Greenspan, "Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition", (Ph.D. Thesis), CSRG-155, Dept. of Comp. Sci. U. of Toronto, March, 1984.

[16] S. J. Greenspan, Alexander Borgida, and John Mylopoulos, "A Requirements Modeling Language and its Logic", *Information Systems*, Vol. 11, No. 1, Pergammon Press, 1986, pp. 9-23.

[17] J.V. Guttag, J.J. Horning, J.M. Wing, "The Larch Family of Specification Languages", *IEEE Software Magazine*, September, 1985.

[18] N. Haas, G.G. Hendrix, "An Approach to Acquiring and Applying Knowledge", *Proc. of First National Conf. on Artificial Intelligence*, Stanford Univ., 1980, pp. 235-239, August, 1980.

[19] M.T. Harandi and F.H. Young, "Template Based Specification and Design", *Proc. of Third Int. Workshop on Software Specification and Design*, London, UK, August, 1985, pp. 94-97.

[20] IEEE Guide to Software Requirements Specifications, ANSI/IEEE Std 830-1984, July, 1984.

[21] R.A. Kemmerer, "Testing Formal Specifications to Detect Design Errors", *IEEE Trans. on Software Eng.*, Vol. 11, No. 1, January, 1985.

[22] J.Z. Lavi, "Improving the Embedded Computer Systems Software Process Using a Generic Model", *Proc. of Third Int. Workshop on Software Specification and Design*, London, UK, August, 1985, pp. 127-129.

[23] D.A. McAllester, "Reasoning Utility Package User's Manual", MIT/AIM-667, April, 1982.

[24] A. Mili, *Proc. of Third Int. Workshop on Software Specification and Design*, London, UK, August, 1985.

[25] M.L. Minsky, "A Framework for Representing Knowledge", in *The Psychology of Computer Vision*, P.H. Winston (ed.), McGraw Hill, 1975.

[26] M.L. Minsky, *Society of Mind*, to appear.

[27] J.M. Neighbors, "The Draco Approach to Constructing Software from Reusable Components", *IEEE Trans. on Software Eng.*, Vol. 10, No. 5, Sept. 1984, pp. 564-574.

[28] P.G. Politakis, "Using Empirical Analysis To Refine Expert System Knowledge Bases", (Ph.D. Thesis), CBM-TR-130, Lab. for Comp. Sci. Res., Rutgers U., 1982.

[29] Sanders Associates, "Knowledge Based Requirements Assistant: Interim Technical Report", 1986.

[30] R. Schank, "A Conceptual Dependency Representation for a Computer-Oriented Semantics", Stanford AIM-83, 1969.

[31] E.Y. Shapiro, *Algorithmic Program Debugging*, MIT Press, 1983.

[32] G.J. Sussman, "The Virtuous Nature of Bugs", *Proc. of Conf. on Artificial Intelligence and the Simulation of Behavior*, U. of Sussex, July 1974..

[33] W. Swartout, "The GIST Behavior Explainer", *Proc. of the Third National Conference on Artificial Intelligence*, Washington, DC, August, 1983, pp. 402-407.

[34] Western Institute of Software Engineering, "Using the WISDM Team Method to Define System Requirements", 1986.

[35] P. Zave, "Executable Requirements for Embedded Systems", *5th Int. Conf. on Software Eng.*, San Diego, Cal., March, 1981, San Diego, CA, March, 1981.

Programmer's Apprentice Publications

[36] D. Brotsky, "An Algorithm for Parsing Flow Graphs", (M.S. Thesis), MIT/AI/TR-704, March, 1984.

[37] D. Chapman, "A Program Testing Assistant", *Comm. of the ACM*, Vol. 25, No. 9, September, 1982, pp. 625-634.

[38] D. Chapman, "Cognitive Cliches", MIT/AI/WP-286, April, 1986.

[39] E.C. Ciccarelli, "Presentation Based User Interfaces", (Ph.D. Thesis), MIT/AI/TR-794, August, 1984.

[40] D.S. Cyphers, "Automated Program Explanation", MIT/AI/WP-237, August 1982.

[41] R. Duffey, II, "Formalizing the Expertise of the Assembler Language Programmer", (M.S. proposal), MIT/AI/WP-203, September, 1980.

[42] G. Faust, "Semiautomatic Translation of COBOL into HIBOL", (M.S. Thesis), MIT/LCS/TR-256, March, 1981.

[43] Y.A. Feldman and C. Rich, "Reasoning with Simplifying Assumptions: A Methodology and Example", *Proc. of the Fifth National Conference on Artificial Intelligence*, Philadelphia, PA, August, 1986.

[44] C. Frank, "A Step Towards Automatic Documentation", MIT/AI/WP-213, December, 1980.

[45] S.M. Levitin, "Toward a Richer Language for Describing Software Errors", (B.S. Thesis), MIT/AI/WP-270, June, 1985.

[46] K.M. Pitman, "Interfacing to the Programmer's Apprentice", MIT/AI/WP-244, February, 1983.

[47] H.B. Reubenstein, "A Requirements Analyst's Apprentice: A Proposal", Ph.D. Thesis Proposal, MIT Dept. of Elec. Eng. and Computer Sci., 1986 (*in preparation*).

[48] C. Rich, "Inspection Methods in Programming", MIT/AI/TR-604, (Ph.D. thesis), June, 1981.

[49] C. Rich, "A Formal Representation for Plans in the Programmer's Apprentice", *Proc. of 7th Int. Joint Conf. on Artificial Intelligence*, Vancouver, Canada, August, 1981, pp. 1044-1052.

[50] C. Rich and R.C. Waters, "Abstraction, Inspection and Debugging in Programming", MIT/AIM-634, June, 1981.

[51] C. Rich, "Knowledge Representation Languages and Predicate Calculus: How to Have Your Cake and Eat It Too", *Proc. of Second National Conf. on Artificial Intelligence*, Pittsburgh, PA, August, 1982.

[52] C. Rich and R.C. Waters, "Formalizing Reusable Software Components", *Proceedings of the ITT Workshop on Reusability in Programming*, Newport, RI, September, 1983..

[53] C. Rich, "The Layered Architecture of a System for Reasoning about Programs", *Proc. of the 9th Int. Joint Conf. on Artificial Intelligence*, Los Angeles, CA, August, 1985, pp. 540-546.

[54] D. Shapiro, "Sniffer: a System that Understands Bugs", (M.S. Thesis), MIT/AIM-638, June, 1981.

[55] H.E. Shrobe, "Explicit Control of Reasoning in the Programmer's Apprentice", *Proc. of 4th Int. Conf. on Automated Deduction*, February, 1979.

[56] H.E. Shrobe, "Dependency Directed Reasoning for Complex Program Understanding", (Ph.D. Thesis), MIT/AI/TR-503, April, 1979.

[57] H.E. Shrobe "Common-Sense Reasoning About Side Effects to Complex Data Structures". *Proc. of 6th Int. Joint Conf. on Artificial Intelligence*, Tokyo, Japan, August 1979.

[58] B.K. Steele, "An Accountable Source-to-Source Transformation System", (M.S. Thesis), MIT/AI/TR-636, June 1981.

[59] E.K. Turrisi, "Chapter and Verse Program Description", MIT/AI/WP-256, (B.S. Thesis), June, 1984.

[60] R.C. Waters, "Automatic Analysis of the Logical Structure of Programs", MIT/AI/TR-492, (Ph.D. Thesis), December, 1978.

[61] R.C. Waters, "A Method for Analyzing Loop Programs", *IEEE Trans. on Software Eng.*, Vol. SE-5, No. 3, May 1979, pp. 237-247.

[62] R.C. Waters, "The Programmer's Apprentice: A Session with KBEmacs", *IEEE Trans. on Software Eng.*, Vol. 11, No. 11, pp. 1296-1320, November, 1985.

[63] R.C. Waters, "KBEmacs: A Step Toward the Programmer's Apprentice", MIT/AI/TR-753, May, 1985

[64] R.C. Waters, "KBEmacs: Where's the AI?", *AI Magazine*, Vol. 7, No. 1, Spring 1986.

[65] R.C. Waters, "Program Translation via Abstraction and Reimplementation", *IEEE Trans. on Software Eng., to appear.*

[66] J. Wertheimer, "A Library of Programming Knowledge for Implementing Rule Systems", (M.S. Thesis), Elec. Eng. and Comp. Sci. Dept., Mass. Inst. of Tech., 1986. *to appear*

[67] L.M. Zelinka, "An Empirical Study of Program Modification Histories", MIT/AI/WP-240, February, 1983.

[68] L.M. Zelinka, "Automated Program Recognition", MIT/AI/TR-904, (M.S. Thesis), August, 1986.